

Continuous Top-k Queries over Real-Time Web Streams

Nelly Vouzoukidou

Sorbonne Universits, UPMC Univ Paris 06, UMR 7606, LIP6
Paris, France
nelly.vouzoukidou@lip6.fr

Bernd Amann

Sorbonne Universits, UPMC Univ Paris 06, UMR 7606, LIP6
Paris, France
bernd.amann@lip6.fr

Vassilis Christophides

INRIA Paris, LINC
Paris, France
vassilis.christophides@inria.fr

October 21, 2016

Abstract

The Web has become a large-scale real-time information system forcing us to revise both how to effectively assess relevance of information for a user and how to efficiently implement information retrieval and dissemination functionality. To increase information relevance, Real-time Web applications such as Twitter and Facebook, extend content and social-graph relevance scores with “real-time” user generated events (e.g. re-tweets, replies, likes). To accommodate high arrival rates of information items and user events we explore a publish/subscribe paradigm in which we index queries and update on the fly their results each time a new item and relevant events arrive. In this setting, we need to process *continuous top-k text queries* combining both *static* and *dynamic* scores. To the best of our knowledge, this is the first work addressing how *non-predictable*, dynamic scores can be handled in a continuous top-*k* query setting.

1 Introduction

The proliferation of social media platforms and mobile technology enables people to almost instantly produce and consume information world-wide. In the *real-time Web*, traditional scoring functions based on content similarity or link graph centrality are no longer sufficient to assess *relevance* of information to user needs. For example, textual *content* relevance may severely be obscured by the “churn” observed in the term distribution of real-time media news or social content and those found in historical query logs [20], while *social relevance* [17] of short living information cannot be reliably computed on static graphs. Streams of user events like “replies” (for posting comments), “likes” (for rating content) or “retweets” (for diffusing information)

represent nowadays valuable social feedback [18] on web content¹ that is not yet exploited online to assess *contextual* relevance aspects.

Real-time Web search engines are essentially confronted with a double challenge. First, “ephemeral” Web content [12] should become searchable for millions of users immediately after being published. Second, *dynamic* relevance of streaming information (e.g. *user attention* [28, 21], *post credibility* [14], *information recency* [10]) should be continuously assessed as users provide their feedback.

To tackle both content and score dynamicity, a first solution consists in extending a content retrieval model (textual, spatial) and implementing adequate index structures and refresh strategies for *low-latency* and *high throughput snapshot query evaluation*. Best-effort refresh strategies [7, 16] could determine optimal re-evaluation of active user queries which combined with real-time content indexing [3, 2, 31, 22, 19] can achieve high result freshness and completeness. However, real-time content indexing systems usually accommodate high arrival rates of items at the expense of result accuracy by either (a) excluding a significant portion of the incoming items (e.g. with infrequent keywords) from the index to reduce update costs or (b) by ranking items only by arrival time to support append-only index insertion and thus ignore content relevance to queries [31]. Coupling both *static* (e.g. content similarity to user queries) with *dynamic* aspects of relevance beyond information recency [10] is clearly an open issue in existing real-time search engines.

An alternative solution is a publish/subscribe architecture [15, 25, 27, 26, 4, 13, 5] in which user (*continuous*) queries rather than information items are indexed while their results are updated on the fly as new items arrive. In *predicate-based* [4, 13] publish/subscribe systems, incoming items that match the filtering predicates are simply added to the result list of continuous queries, while in *similarity-based top-k* [15, 25, 27, 26, 5] systems, matching items have also to exhibit better relevance w.r.t. the items already appearing as the top-k results of continuous queries. In publish/subscribe systems both (a) *early pruning of the query index traversal* for locating relevant queries to an incoming item and (b) the *efficient maintenance of their results lists* are challenging problems. Only recently, these problems have been studied for more dynamic settings such as decaying information relevance as time passes (e.g. for textual [26, 27] or spatio-textual [5] streams). However, there is an important difference between information decay and social feedback: while information freshness is defined as a *known in advance, global* function applied on all item scores simultaneously, social feedback is *locally* defined as a stream of non-predictable, random events per item. First, such randomness introduces a new degree of freedom requiring adaptive optimization strategies to the incoming feedback events. Second, the item score dynamics significantly varies and calls for new techniques to balance processing cost (memory/CPU) between items with high and low feedback.

In this paper we are studying a new class of continuous queries featuring *real-time scoring functions* under the form of *time decaying positive user feedback* for millions of social feedback events per minute and millions of user queries. In a nutshell, the main contributions of our work are:

- We introduce continuous top- k queries over real-time web streams aggregating both *static item content* and *dynamic real-time event* scores. Then, we decompose the continuous top- k query processing problem into *two separate tasks* matching item contents and feedback events.
- We propose new *efficient in-memory data structures* for indexing continuous top- k queries

¹This trend is expected to be amplified in the future Internet of Things (IoT) where users can provide online feedback regarding almost any digital or physical object [8].

when matching incoming user events along with a family of *adaptive algorithms for maintaining the result of top- k queries with highly dynamic scores*.

- We experimentally evaluate our index structures over a real-world dataset of 23 million tweets collected during a 5-month period and compare the impact of dynamic scores in the performance of our algorithms and associated data structures.

The rest of the paper is organized as follows. Section 2 gives a formal definition of the problem. Section 3 presents our proposed solution and pruning techniques for event matching. Then, Section 4 describes a number of index implementations that apply these techniques whose experimental evaluation is detailed in Section 5. Related work is presented in Section 6 and the main conclusions drawn from our work are summarized in Section 7.

2 Real-time top- k queries

In this section we formally define the problem of evaluating continuous top- k queries with dynamic scores.

2.1 Data Model

The general data model builds on a set of search queries Q , a set of items I and a set of events E . Each feedback event $e \in E$ (e.g., “replies”, “likes”, “retweets”) concerns exactly one target item i denoted by $target(e)$ (e.g., media news articles, social posts). The set of all events with target i is denoted by $\mathcal{E}(i)$. We also assume a time-stamping function $ts : Q \cup I \cup E \rightarrow \mathcal{T}$ which annotates each query, item and event with their arrival time-stamp in the system. This function introduces a weak ordering between queries, items and events and formally transforms all sets Q , I and E into streams Q^{ts} , I^{ts} and E^{ts} .

Continuous top- k queries A continuous top- k query $q = (k, \mathcal{S}_{qu}, \mathcal{S}_{dyn}, \alpha)$ is defined by a positive constant k , a static query score function \mathcal{S}_{qu} , an event score function \mathcal{S}_{ev} and some non negative query score weight $\alpha \leq 1$:

- The constant k defines the maximum *size* of the query result at each time instance.
- The *query score* function $\mathcal{S}_{qu} : Q \times I \rightarrow [0, 1]$ returns a static score of query $q \in Q$ and item $i \in I$. It may capture popular content similarity measures (e.g. textual like cosine, Okapi BM25 or spatial similarity), but it might also reflect other static, query independent scores like source authority [24] or media focus [23].
- The *event score* function $\mathcal{S}_{ev} : E \rightarrow [0, 1]$ returns a positive score for each $e \in E$. Events can be scored according to their *type and content* (e.g. comment, share, click, like) [11] as well as their *author* (e.g. friend versus anonymous [1]). Distance measures between the author of an even the user subscribing a query could be also considered [19].
- The *dynamic (feedback) score* function $\mathcal{S}_{dyn} : I \times \mathcal{T} \rightarrow [0, \infty]$ aggregates the scores of all events $e \in \mathcal{E}(i)$ with target i up to time instant τ :

$$\mathcal{S}_{dyn}(i, \tau) = \sum_{e \in \mathcal{E}(i), ts(i) \leq ts(e) \leq \tau} \mathcal{S}_{ev}(e)$$

Observe that we only allow positive event scores and the aggregated event score is unbound.

- The *total score* of some item i w.r.t. some query q is defined a linear combination of the query score $\mathcal{S}_{qu}(q, i)$:

$$\mathcal{S}_{tot}(q, i, \tau) = \alpha \cdot \mathcal{S}_{qu}(q, i) + (1 - \alpha) \cdot \mathcal{S}_{dyn}(i, \tau)$$

Observe that feedback on items is ignored when $\alpha = 1$ whereas items are only ranked with respect to their feedback (independently from the query score) when $\alpha = 0$. As a matter of fact, we consider a general form of *static* and *dynamic* scoring function that abstracts several score aspects of items proposed in the literature.

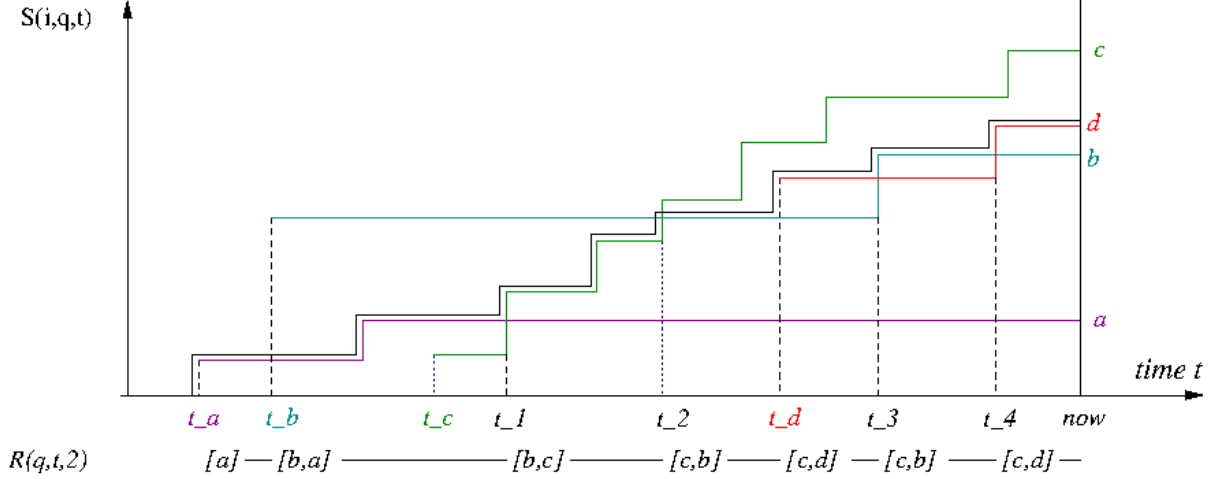


Figure 1: Top-2 result evolution of query q

Example 1 Figure 1 illustrates the high dynamicity of the query result of a single continuous top- k query over real-time web streams. It shows the evolution of the top-2 result of some q for item set $I^{ts} = \{a, b, c, d\}$. The score evolution of each item is represented by a stepwise line where each monotonic increase corresponds to the arrival of some event (aggregated event score). The minimum score of q is represented by a bold line. Each item has an initial positive query score and the query result is updated seven times: at the arrival of a new item (item match at τ_a , τ_b and τ_d) as well as at the arrival of a new event (event match at τ_1 , τ_2 , τ_3 and τ_4). Observe that the arrival of a new item does not necessarily trigger a query result update (e.g. arrival of c) and an item can disappear and reappear in the result (item b disappears at τ_d and reappears at τ_3).

Decay function To take account of information freshness [10], we consider an order-preserving decay function $decay(s, d)$ which can be applied to query, item or event score values s : given a time duration d such that $decay(s, 0) = s$ and $decay(s, d) \leq decay(s, d')$ for all $d > d'$, i.e. if $s > s'$, then $decay(s, d) > decay(s', d)$ for any duration d . Order-preserving decay functions² allow in particular the use of the backward decay techniques [9] where all scores are computed, stored and compared with respect to some *fixed reference time instant* τ_0 used as a landmark. It should be stressed that two possible semantics for event decay can be defined. The *aggregated decay semantics* consists in applying decay to the aggregated event score with respect to the target item age, whereas the *event decay semantics* consists in applying decay to each individual event with respect to the event age. In the first case, all events have the same decay, whereas

²Whereas any linear or exponential function is order-preserving, it is possible to define polynomial functions which are not order preserving.

the second case favors recent events to old ones. In the special case of linear decay both semantics lead to the same results. The aggregated decay semantics maintains the order preserving property, whereas the event decay semantics might change the item order for order preserving exponential decay. In the following and in our experiments, we assume a linear order-preserving decay function.

Formal semantics and problem statement We denote by $\mathcal{P}(q)$ the set of *relevant* items which have a strictly positive query score $\mathcal{S}_{qu}(q, i) > 0$. The *top-k result* of a continuous query $q = (k, \mathcal{S}_{qu}, \mathcal{S}_{dyn}, \alpha)$ at some time instant τ , denoted $R(q, \tau, k)$, contains the subset of maximally k *relevant* items $i \in \mathcal{P}(q)$ with the highest total scores $\mathcal{S}_{tot}(q, i, \tau)$. In other words, for each item $i \in R(q, \tau, k)$ there exists no other (relevant) item $i' \in I - R(q, \tau, k)$ with a higher total score $\mathcal{S}_{tot}(q, i', \tau) > \mathcal{S}_{tot}(q, i, \tau)$. Using the backward decay technique described before, we suppose that all scores are computed, stored and compared with respect to some *fixed reference time instant* τ_0 used as a landmark.

We can now state the general real-time top- k query evaluation problem which we will address in the rest of this paper:

Problem statement 1 *Given a query stream Q^{ts} , an item stream I^{ts} , an event stream E^{ts} and a total score function \mathcal{S}_{tot} (with order-preserving decay), maintain for each query $q \in Q^{ts}$ its continuous top- k result $R(q, \tau, k)$ at any time instant $\tau \geq ts(q)$.*

2.2 Query execution model

This top- k query maintenance problem can be reformulated into a sequence of updates triggered by the arrival of new queries, items and events at different time-instants. We will consider a fixed time instant τ and denote by

- $\mathcal{Q}(i) = \{q | i \in R(q, \tau, k)\}$, the set of *active queries* where item i is published at time instant τ ;
- $\mathcal{C}(q) = \{i | i \notin R(q, \tau, k) \wedge \mathcal{S}_{qu}(q, i) > 0\}$, the set of all *candidate items* which might be inserted into the result of q on a later time instant $\tau' > \tau$ due to feedback score updates.
- $\mathcal{C}(i) = \{q | i \in \mathcal{C}(q)\}$, the set of all *candidate queries* for item i ;

Example 2 *For the example of Figure 1, during time interval $(\tau_2, \tau_d]$, query q appears in the active sets of items c and b . After the arrival of item d , query q is inserted into the active set of d and moves from the active set to the candidate set of b . The arrival of a new event targeting b at time instant τ_3 switches q from the active (candidate) set to the candidate (active) set of d (b).*

Problem statement 1 can be decomposed into three separate matching tasks at a given time instant τ :

Query matching: Given a query q , compute the set $R(q, \tau, k)^+$ of items that should be added to the top- k result of q ;

Item matching: Given an item i , identify the set $\mathcal{Q}(i)^+$ of all queries that should be updated by adding i in their result.

Event matching: Given an event e with target item $i = target(e)$, identify the set $\mathcal{Q}(i)^+$ of all queries that should be updated by adding i to their result.

Observe that through the definitions of $\mathcal{Q}(i)$ and $R(q, \tau, k)$ all three matching tasks are strongly related and some tasks could be partially solved by using others. For example, item matching can be solved by re-evaluating (refreshing the result of) all queries while event matching can be solved by re-executing the item matching task for its target item with the updated event score. We will show in Section 3 and experimentally verify in Section 5, this solution exhibits serious performance limitations for information items with highly dynamic scores. In the rest of this article we are mainly interested in the definition and implementation of an efficient solution for the event matching task by relying on existing efficient solutions for the query and item matching task.

Query evaluation architecture and algorithm As a matter of fact, by considering distinct matching tasks, we can devise a modular architecture and experiment with different data structures and algorithms. As depicted in Figure 2 the main processing modules Query Handler (QH), Item Handler (IH) and Event Handler (EH) are independent and share data structures for indexing queries (Query Index) and items (Item Index). Events are not stored but dynamically aggregated in the corresponding item scores and updated in the Item Index.

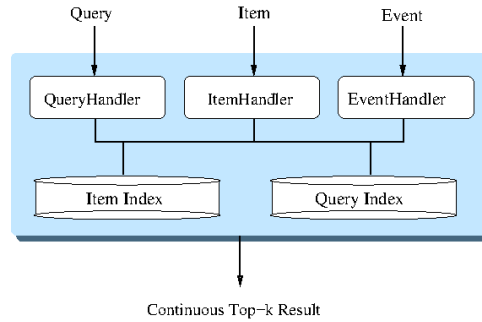


Figure 2: Architecture

The general interaction of these models for processing incoming queries, events and items is illustrated in Algorithm 1. Function `QH.processQuery()` indexes each new incoming query q . Functions `IH.processItem()` and `EH.processEvent()` match new incoming items and events against the registered queries and identify the queries whose result (top- k set) have to be updated. Function `QI.matchItem` computes $\mathcal{Q}(i)^+$ for a given item (contents) i with a positive constant event score. This supposes that the Item Handler supports continuous top- k queries with *inhomogeneous* score functions aggregating a *query-dependent score* (e.g. cosine similarity) and a *query-independent* constant value (for example item popularity). Such an index structure has been proposed for example in [27].

Function `EH.matchEvent()` matches each new incoming event and generates $\mathcal{Q}(\text{target}(e))^+$. In the rest of this paper we are mainly interested in the efficient implementation of this function and our approach will be described in Section 3. Function `RTS.add()` publishes item i in the result of q . It also updates the publication and candidate sets for i and query q and, if necessary, for item i' which has been replaced by i in the result of q (see Example 2).

Theorem 1 *Algorithm 1 guarantees that the results of all queries are correct.*

Proof: [sketch] We assume that function `II.matchQuery`, `QI.matchItem` and `EH.matchEvent` correctly compute the update sets $R(q, \tau, k)^+$ and $\mathcal{Q}(i)^+$, respectively, as defined in Section 2. The correctness of the item matching function directly implies that all items are coherently *added* to the query results by function `RTS.add`. We then have to show that an item can only

Algorithm 1: Real-time top-k query evaluation

```
1 QH.processQuery( $q$ : Query)
2   |   QI.addQuery ( $q$ );
3   |   II.matchQuery ( $q$ );
4   |   foreach  $i \in R(q, \tau, k)$  do
5   |   |   RTS.add ( $q, i$ );
6 IH.processItem( $i$ : Item)
7   |   II.addItem ( $i$ );
8   |   QI.matchItem ( $i$ );
9   |   foreach  $q \in \mathcal{Q}(i)^+$  do
10  |   |   RTS.add ( $q, i$ );
11 EH.processEvent( $e$ : Event)
12  |   EH.matchEvent ( $e$ );
13  |   foreach  $q \in \mathcal{Q}(\text{target}(e))^+$  do
14  |   |   RTS.add ( $q, i$ );
15 II.matchQuery( $q$ : Query)
16  |   compute  $R(q, \tau, k)^+$ ;
17 QI.matchItem( $i$ : Item)
18  |   compute  $\mathcal{Q}(i)^+$ ;
19 EH.matchEvent( $e$ : Event)
20  |   compute  $\mathcal{Q}(\text{target}(e))^+$ ; // see Section 3
21 RTS.add( $q$ : Query,  $i$ : item)
22  |   remove top- $k$  item (if it exists) from the result of  $q$ ;
23  |   add item  $i$  to the result of query  $q$ ;
```

be removed from the query result by being replaced by an item with a higher score (function `RTS.add`). This can easily be shown by the fact that we allow only positive event scores (\mathcal{S}_{tot} is monotonically increasing). Observe that with negative event scores, we would have to define a new function `RTS.del` which allows to replace some item after one or several negative scores and replace them by some other candidate item.

3 Event Handler algorithms

In this section we will present two algorithms for solving the (positive and negative) event matching problems described in the previous section.

The first algorithm is called *All Refresh (AR)*. The basic idea is to increase for each new positive event e the query-independent score of its target item $target(e)$ and to retrieve the queries whose result have to be updated, by re-evaluating the item in the Item Handler.

Algorithm 2: The *AR* algorithm

```

1 EH.matchEventSimple( $e : Event$ )
2   if  $\mathcal{S}_{ev}(e) == 0$  then
3     return  $\emptyset$ ;
4    $i := target(e)$ ;
5    $dynscore(i) := dynscore(i) + score(e)$ ;
6   II.updateItem( $i$ );
7   QI.matchItem( $i$ );
8   for  $q \in Q(i)^+$  do
9     RTS.add( $q, i$ );
```

Theorem 2 *Algorithm AR is correct.*

Proof: [sketch] We assume that function `II.matchQuery`, `QI.matchItem` and `RTS.add` are correct and condition $AlgCorr(i)$ holds for target item i before the event. We show that for a given item i there exists no query $q \in \mathcal{C}(i)$ where $\mathcal{S}_{tot}(q, i) > \mathcal{S}_{min}(q)$ after the execution of `EH.processEvent(e)` processing an event e with target item i . `EH.processEvent(e)` calls `QI.matchItem` (through function `EH.matchEventSimple`) and then `RTS.add(i)` after the update of the dynamic score. By definition, the item matching function `QI.matchItem` returns all queries where $\mathcal{S}_{tot}(q, i) > \mathcal{S}_{min}(q)$ and after processing `RTS.add(i)`, $Q(i)$ contains all these queries.

The *AR* algorithm achieves an acceptable performance when events are quite rare, compared to the item's arrival rate, but becomes inefficient on a real-time web system such as Twitter with feedback events arriving from millions of users. In a highly dynamic setting we would expect that a single event on an item, e.g. a single retweet or favorite, should have, on average, a small impact on the set of queries that would receive it. However, by re-evaluating the item through the Item Handler, we would have to re-compute a potentially long list of candidate queries that have already received i .

We propose the *Real Real-Time Search (R^2TS)* algorithm, which is based on the idea of using the Item Handler for computing for each item a list of query candidates that could potentially be updated by any future event. These lists are then processed by the Event Handler for retrieving all queries whose result has to be updated with the item targeted by the incoming events. Compared to *AR*, *R^2TS* avoids examining the same list of candidate queries for an

item i each time a new event arrives targeting i . The efficiency of this algorithm relies on the following observations:

- each item i has a partial set of query candidates q which might be updated by future events;
- each event increases the score of a single item and triggers a limited number of updates
- there exists a maximal dynamic aggregated score $\theta(i)^{max}$ for each item which depends on the scoring function and the number of events it will receive.

The total scoring function (Equation 1) linearly increases the item score for each new event. This leads to a straightforward way of choosing the candidates by setting a threshold $\theta(i) > 0$ for each item i which controls the number of query candidates and which is bound by the maximal dynamic score i can receive by all future events.

Candidates and threshold condition: Given a positive threshold value θ , the candidates $|\mathcal{C}(i, \theta(i))| \subseteq Q$ of an item i is the set of queries: $|\mathcal{C}(i, \theta(i))| = \{q \in Q | \mathcal{S}_{min}(q) \in (S(q, i), S(q, i) + \gamma \cdot \theta]\}$. We say that the *threshold condition* $T(e, \theta)$ holds for some event e iff the dynamic score of its target item does not exceed threshold θ : $T(e, \theta) = (\mathcal{S}_{dyn}(target(e)) + \mathcal{S}_{ev}(e)) < \theta$. It is easy to see that when $T(e, \theta)$ is true, then all queries updated by e are in the candidate set $|\mathcal{C}(i, \theta(i))|$. Otherwise, the candidate set is no longer valid and needs to be recalculated through the Item Handler by increasing threshold θ .

The threshold-based approach of R^2TS is detailed in Algorithm 3. Each new item i is assigned with a constant positive threshold $\theta(i)$ (line 6) and a refresh counter $r(i)$ (line 7). $\theta = r(i) \cdot \theta(i)$ represents the aggregate threshold value after $r(i)$ refresh steps. When a new event arrives, function `EH.matchEvent` first increases the dynamic score (line 12) and updates the item index. If the threshold condition does not hold, the candidate list is updated by calling procedure `EH.refreshCandidates`. The instructions from line 23 to line 26 increment the refresh counter and compute the query candidates by calling the item handler with the new aggregated threshold (item c is a placeholder copy of item i with a virtual aggregated event score corresponding to the threshold of i before the next refresh). Observe that for $\theta(i) = 0$, $\mathcal{C}(i)$ contains *exactly* all queries that have to be updated by the new event. At line 16 the threshold condition holds for event e and `EH.matchEvent` copies all candidate queries q where the minimum score of q is strictly smaller than the global score of i to the update set $\mathcal{U}(e)$ (lines 16 to 19).

Theorem 3 *Algorithm R^2TS is correct for positive event scores.*

Proof: [sketch] We have to show that both functions, `IH.processItem` and `EH.matchEvent`, guarantee that for all items i and all queries $q \in Q$, if $\mathcal{S}_{tot}(q, i) > \mathcal{S}_{min}(q)$, then $q \in \mathcal{Q}(i)$ (condition *AlgCorr*). It is easy to see that the correctness condition holds for some new item i after the execution of function `IH.processItem` (proof of Theorem 1).

We now show that *AlgCorr* also holds after the execution of function `EH.matchEvent`. If $\mathcal{S}_{ev}(e) = 0$ the result does not change (line 9). Otherwise, lines 11 to 13 update the event score of target item $target(e)$. Line 14 checks if the candidate set $|\mathcal{C}(i, \theta(i))|$ of item i has to be updated. If this is the case, lines 23 to 26 increment $r(i)$ by one and recompute the candidate set $|\mathcal{C}(i, \theta(i))|$. We know at line 16 that the candidate set $|\mathcal{C}(i, \theta(i))|$ of i contains *all* queries $q \in Q - \mathcal{Q}(i)$ where

$$\mathcal{S}_{min}(q) \leq \mathcal{S}_{tot}(q, i) = \alpha \cdot \mathcal{S}_{qu}(i, q) + (1 - \alpha) \cdot r(i) \cdot \theta(i) \quad (1)$$

Algorithm 3: The R^2T S algorithm

```

1 IH.processItem( $i$ : Item)
2   II.addItem( $i$ );
3   QI.matchItem( $i$ );
4   for  $q \in \mathcal{Q}(i)^+$  do
5     RTS.add( $q, i$ );
6    $\theta(i) := \text{EH.initThreshold}$ ;
7    $r(i) := 0$ ;
8 EH.matchEvent( $e$ : Event): set(Query)
9   if  $\mathcal{S}_{ev}(e) == \emptyset$  then
10    return  $\emptyset$ ;
11    $i := \text{target}(e)$ ;
12    $\mathcal{S}_{dyn}(i) := \mathcal{S}_{dyn}(i) + \mathcal{S}_{ev}(e)$ ;
13   II.updateItem( $i$ );
14   if  $\mathcal{S}_{dyn}(i, \tau) > r(i) \cdot \theta(i)$  then
15     EH.refreshCandidates( $i$ );
16   for  $q \in |\mathcal{C}(i, \theta(i))|$  do
17     for  $i' \in R(q, \tau, k)$  do
18       if  $\mathcal{S}_{tot}(q, i) > \mathcal{S}_{tot}(q, i')$  then
19         RTS.add( $q, i$ );
20 EH.refreshCandidates( $i$ : Item)
21    $c := \text{copy}(i)$ ;
22   if  $\theta(i) > 0$  then
23      $r(i) := r(i) + \lfloor \mathcal{S}_{dyn}(i, \tau) / \theta(i) \rfloor + 1$ ;
24      $\mathcal{S}_{dyn}(c, \tau) := r(i) \cdot \theta(i)$ ;
25   QI.matchItem( $c$ );
26    $\mathcal{C}(i) := \mathcal{Q}(c)^+ - \mathcal{Q}(i)$ ;

```

Then we can show that $|\mathcal{C}(i, \theta(i))|$ contains all queries that have to be updated by event e (and other queries which are not updated). By definition and equation 1, for all queries q' to be updated by event e the following holds

$$\begin{aligned}
\mathcal{S}_{min}(q') < \mathcal{S}_{tot}(q', i) &= \mathcal{S}_{stat}(q', i) + \gamma \cdot \mathcal{S}_{dyn}(i, \tau) \\
&\leq \mathcal{S}_{stat}(q', i) + \gamma \cdot r(i) \cdot \theta(i)
\end{aligned}$$

which means that $q' \in |\mathcal{C}(i, \theta(i))|$ and if $\mathcal{S}_{tot}(q, i) > \mathcal{S}_{min}(q)$, then $q \in \mathcal{Q}(i)$ after executing lines 16 to 19 (condition *AlgCorr*).

The challenge arising from this algorithm is twofold. Identify optimal threshold values $\theta(i)$ for each item and index the obtained candidate queries in a way that allows efficient retrieval of updates during the event matching operation (see Section 4).

Cost analysis: The choice of threshold $\theta(i)$ controls the number of candidate query refresh operations and has an important impact on the overall performance of the system.

From Algorithm 3 we can easily understand that higher values of $\theta(i)$ minimize the number of costly candidate refresh operations (by calling the Item Handler) but also might generate a large number of false positive query candidates which will never be updated by an event targeting i .

On the other hand, low values of $\theta(i)$ lead to more frequent costly candidate re-evaluations on the Item Handler.

In the following we will use the notation in Table 1. Under the assumption that each event

i	target item
$\theta(i)$	estimated threshold value for i
$\theta(i)^{max}$	estimated maximal aggregated event score for i
N_i	estimated total number of events for i
$r(\theta(i))$	estimated number of candidate refresh operations for i
$costIH(i, \theta(i))$	aggregated <i>item matching cost</i>
$costEH(i, \theta(i))$	aggregated <i>event matching cost</i>
$ \mathcal{C}(i, \theta(i)) $	average size of the candidate list of item i .
$costT$	cost of checking if item i updates candidate q
$costM(i)$	average item matching cost for i
$costC(i, \theta(i))$	average candidate list construction cost for i and $\theta(i)$
$cost(i, \theta(i))$	aggregated <i>total matching cost</i>

Table 1: Cost function parameters

targets only one item and that items are processed independently, we are interested in finding an optimal value for the threshold value $\theta(i)$ that minimizes the aggregated total cost $cost(i, \theta(i))$ for each item i . This local optimization leads to a globally optimized evaluation cost.

By definition the total aggregated matching cost is defined by the sum of the aggregated item and the aggregated event matching costs:

$$cost(i, \theta(i)) = costIH(i, \theta(i)) + costEH(i, \theta(i))$$

If we suppose that all events have the same event score $\mathcal{S}_{ev}(e)$, then $r(\theta(i)) = \lceil \theta(i)^{max} / \theta(i) \rceil$.

We can estimate the aggregated item and event matching costs $costIH(i, \theta(i))$ and $costEH(i, \theta(i))$ as follows (Algorithm 3). First, each candidate list is refreshed $r(\theta(i))$ times where each refresh consists in matching the item and constructing the candidate list (function `EH.refreshCandidates`):

$$costIH(i, \theta(i)) = r(\theta(i)) \cdot (costM(i) + costC(i, \theta(i))) \quad (2)$$

Second, each event is matched in the Event Handler N_i times. Supposing that there is *no early stopping condition* when checking the candidates, the cost of every event match operation depends on the average size of the candidate list $|\mathcal{C}(i, \theta(i))|$ and on the (constant) cost $costT$ of checking if item i updates a given query candidate q :

$$costEH(i, \theta(i)) = N_i \cdot |\mathcal{C}(i, \theta(i))| \cdot costT \quad (3)$$

Finding optimal $\theta(i)$: The value of $|\mathcal{C}(i, \theta(i))|$ depends on the distribution of the query minimum scores. Our cost model relies on the assumption that the average size of all candidate lists produced by each refresh step linearly depends on threshold value $\theta(i)$: $|\mathcal{C}(i, \theta(i))| = a \cdot \theta(i)$,

where a is a positive constant³. Similarly, we assume that the candidate creation $costC(i, \theta(i))$ depends linearly on $\theta(i)$: $costC(i, \theta(i)) = b \cdot \theta(i)$. This cost abstraction simplifies the computation of the optimal $\theta(i)$ and they have been also experimentally validated.

Given $r(\theta(i)) = \theta(i)^{max}/\theta(i)$ (we consider here the real value instead of the integer floor value), Equations (2) and (3) the total $cost(i, \theta(i))$ can be written as:

$$cost(i, \theta(i)) = \frac{1}{\theta(i)} \cdot \overbrace{\theta(i)^{max} \cdot costM(i)}^{c_1} + \underbrace{\theta(i) \cdot N_i \cdot a \cdot costT}_{c_2} + \underbrace{\theta(i)^{max} \cdot b}_{c_3}$$

We minimize the $cost$ function using the first derivative:

$$\frac{d(cost(i, \theta(i)))}{d\theta(i)} = -\frac{c_1}{(\theta(i))^2} + c_2$$

Function $cost$ is monotonically decreasing in the interval $(0, \sqrt{c_1/c_2})$ and increasing in $(\sqrt{c_1/c_2}, \theta(i)^{max}]$, thus making $\theta(i) = \sqrt{c_1/c_2}$ the optimal value:

$$\theta(i)^{opt} = \sqrt{\frac{1}{a} \cdot \frac{\theta(i)^{max}}{N_i} \cdot \frac{costM(i)}{costT}}$$

This equation essentially shows that $\theta(i)^{opt}$ depends on the query distribution in the Item Handler index (first factor) and increases with the ratio $costM(i)/costT$ between the item matching cost and the event test cost (third factor).

4 Candidate indexing

The Event Handler matches incoming events against a precomputed list of query candidates which has to be regularly refreshed. In the above cost model, we do not take account of the cost of maintaining the candidate list. This cost clearly depends on the size of the list, as well as, on the choice of the data structures for indexing candidates in the Event Handler. In this section we introduce three indexing schemes aiming to optimize the cost of storing and retrieving candidates. We are particularly interested in finding efficient early stopping conditions during event matching in order to avoid visiting all candidates.

The main task of the Event Handler is to retrieve result updates triggered by any arriving event. Since each event e increases the score of a single item $target(e)$, these updates will only concern a subset of this item's candidate queries. Hence, the building block of all the indexes we propose is a dictionary from each item to its set of candidates. The proposed indexes aim at organizing the *posting lists*, i.e. the candidates sets in a way that decreases the number of false positives encountered during event matching and to efficiently support insertions and deletions. Although deletion in both cases is not necessary for the correctness of the algorithm, maintaining these queries as candidates would lead to an unnecessary number of false positives, deteriorating the overall event-matching performance for following event evaluations. A straightforward implementation of the Event Handler posting lists is to maintain an unordered set of candidates as illustrated in Figure 3-(a). Assuming a dynamic array implementation of this index, insertions and deletions of candidates can be performed in (amortized) constant time.

³A more precise cost model estimating the optimal threshold *for each individual refresh* would need a query/item/event distribution model which is difficult to obtain for a real-world workload.

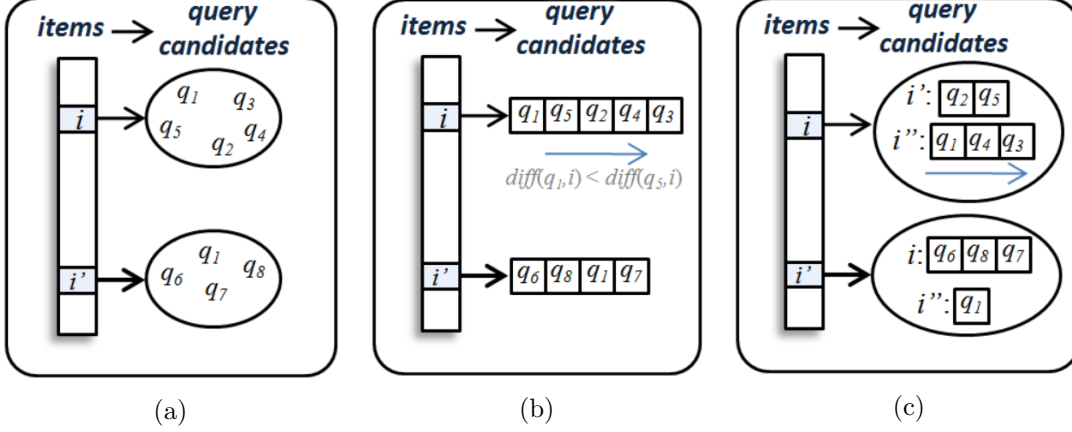


Figure 3: (a)Simple (b)Ordered (c)Partitioning indexes

On the one hand, if $\theta(i)$ is large enough, this solution has the advantage of avoiding candidate list updates for a potentially big number of arriving events (as long as the $\theta(i)$ -condition holds). On the other hand, it is not possible to apply any early stopping condition, and all candidate queries need to be checked for updates.

Following the “sort and prune” principle, we try to define an efficient ordering and appropriate data structures for matching query candidates that will allow us to define an early stopping condition (Figure 3-(b)). As defined in Section 3, a query q is a candidate of an item i if $\text{diff}(q, i) = \mathcal{S}_{\min}(q) - \mathcal{S}(q, i) \in (0, \theta(i)]$ and $\text{diff}(q, i)$ represents the lower dynamic score bound i must receive before updating q ($\mathcal{S}_{\min}(q)$ is monotonically increasing). So, if candidates are ordered by value $\text{diff}(q, i)$, on event arrival, it is sufficient to visit only candidates with *negative* $\text{diff}(q, i)$. Notice that this set of visited candidates guarantees not only zero false negatives but also zero false positives. Nevertheless, maintaining this order in a dynamic environment is a non-trivial problem: from the moment a query becomes a candidate of an item i until to a new event with target i , the minimal scores of other candidates might have changed (increased) due to result updates or, more frequently, due to events received by their k -th item. In the following, we will discuss the implications of maintaining the accurate order and present two lazy re-ordering approaches reducing update costs.

Exhaustive ordering Index: This solution accurately maintains the order of query candidates by the score difference $\text{diff}(q, i)$. To achieve this, it executes all necessary re-orderings on each $\mathcal{S}_{\min}(q)$ change, independently to whether it is due to an event or an item and in all postings of items that q is a candidate. We will observe in our experiments that in a real-world workload the performance gains from avoiding false positives are outperformed by the time wasted to maintain the order.

The following two solutions follow a lazy re-ordering approach, in order to achieve a reasonable trade-off between the cost of eliminating false positives and the cost of re-orderings candidates.

Static-order Index: This solution sorts all candidates only at the creation of the candidate list. New queries are inserted to the beginning of the candidate lists, independently of their $\text{diff}(q, i)$ values. All minimal score $\mathcal{S}_{\min}(q)$ changes are also ignored. On event arrival, the matching algorithm visits all candidate queries for which the *stored* (and not actual) difference score is lower than the item’s dynamic score: $\text{diff}(q, i) < \mathcal{S}_{\text{dyn}}(i, \tau)$. It is easy to show that this condition is safe but also converges with time to the previous simple Event Handler solution.

Lazy re-ordering Index: The two previous approaches either have a very high maintenance

cost due to frequent re-ordering operations (*Exhaustive*), or an inefficient event matching due to lack of maintenance of this order (*Static-order*). The Lazy approach lies in-between by following a lazy “on false-positive” update approach. Given an item i , any order changes caused by minimal score updates are ignored until the next arrival of a new event with target i . The heuristics behind this approach is that many re-orderings of the *Exhaustive* approach never are exploited because candidate queries are moved several times before actually being visited on an event arrival. Therefore, the *Lazy* approach attempts to minimize the number of re-orderings at the expense of introducing false positives on event matching. Given that in the general case the cost of re-ordering an element in a sorted list has a logarithmic complexity, while the cost of eliminating a false positive is constant, the *Lazy* approach is expected to outperform the *Exhaustive* one.

Item Partitioning Event Handler: Ordered indexes reach their limits for highly varying $\text{diff}(q, i)$ values. The *Item Partitioning* approach relies on the following observation: if two queries q_1 and q_2 are both candidates of an item i and both have the same item i' as their k -th result item, both difference score values $\text{diff}(q_1, i)$ and $\text{diff}(q_2, i)$ are synchronized and the relative order of both queries in the posting list of i will remain the same. For this reason, we organize the query candidates of each item, with respect to their k -th element. The order assigned in each such group during insertion remains constant for as long as their k -th result remains the same, i.e. as long as they do not receive any updates. For example, in Figure 3, item i has 5 candidate queries, two of which (q_2 and q_5) have i' as their k -th element. These queries are grouped together (Figure 3-(c)) in a sorted list, (ordered according to diff). For as long as i' remains the k -th element of both these candidate queries, the ordering of the list is static. On event arrival, the matching operation checks each one of these groups if the corresponding item, until reaching true negative candidate. In case of an update, however, of a query q by an item, q has to be re-indexed in the group of its new k -th element in all items where it is a candidate. Despite the additional cost on the query update operation, as we will see later in Section 5, the minimization of both re-orderings (only on the case of updates) and false positives leads to an overall better performance than the previous approaches.

5 Experiments

In this section we experimentally evaluate the algorithms presented in Section 3 and the data structures proposed in Section 4, using a real dataset collected from Twitter’s public streams. Through these experiments we compare the performance of the R^2TS algorithm implementations over the simple candidates (SIMPLE), the lazy ordering (LAZYORDER) and the item partitioned (ITEMPART) index with the naïve approach of the *AR* algorithm (NAIVE) over a number of parameters, like the total number of stored continuous queries, the size k of query result lists, and the weight of user feedback (γ) on the total score function. Additionally, we are interested in assessing the impact of the $\theta(i)$ tuning parameter over the overall system performance. Our experiments rely on a slightly modified version of the item matching algorithm⁴ (Item Handler) presented in [27] to additionally detect candidate queries given a value of $\theta(i)$.

Experiments were conducted using an Intel Core i7-3820 CPU@3.60. Algorithms were implemented in Java 7 and executed using an upper limit of 8GB of main memory (-Xmx8g). The configuration used only one core for the execution of the different algorithms. We present the *average execution times of three identical runs* after an initial warm-up execution. All queries were stored before any item or event evaluation and their insertion time is not included in the results. We collected a real-world dataset from the Twitter Stream API over a period of 5

⁴Available as an open source library `continuous-top-k`: <https://code.google.com/p/continuous-top-k/>

	#items	#events	min	avg
DS1	10 676 097	13 787 349	1	1.29
DS5 (default)	201 581	2 013 427	5	9.99
DS10	56 417	1 105 639	10	19.60

Table 2: Number of items and events in each dataset

parameter	default value	range
#queries	900 000	[100 000, 900 000]
α	0.3	$(1 - \gamma)/2$
β	0.3	$(1 - \gamma)/2$
γ	0.4	[0.05, 0.95]
k	1	[1, 20]
$\theta(i)$ strategy	$\theta(i)^{max}/2$	[0, 0.2] or $[0, \theta(i)^{max}]$

Table 3: Experimental parameters

months (from March to August 2014). In our data model, an original tweet corresponds to an item and a retweet to a feedback signal, i.e. an event for the original item. From this set we have filtered out non-English tweets, as well as those without any retweets (events), leading to a dataset of more than 23 million tweets and retweets (DS1) (Table 2). Two additional subsets were created by considering only tweets (and the corresponding retweets) with at least 5 (DS5) or 10 retweets (DS10) per item. DS5 dataset is the default dataset for the experiments and contains 2.2 million tweets and retweets. Queries were generated by uniformly selecting the most frequent n-grams of 1, 2 or 3 terms from the tweet and retweet dataset leading to an average length of 1.5 terms per query. In each of the following experiments we change one parameter within a given range, while all system parameters remain constant. Table 3 shows the default values for each parameter, as well as the range of each parameter used for the corresponding experiments.

Exact maximal threshold $\theta(i)^{max}$: In this experiment we use as maximal threshold $\theta(i)^{max}$ the exact final aggregated event score $\mathcal{S}_{dyn}(i, \tau)$ (Equation 2.1) of each item i . The horizontal axis of Figure 4a represents the percentage (from 0 to 100%) of the $\theta(i)^{max}$ value assigned as $\theta(i)$ value to an incoming item while the vertical axis represents the time required to match the items and events of the dataset DS5. NAIVE execution time is independent of the $\theta(i)$ value and thus shown as a constant line. Note that in the special case of $\theta(i) = 0$ all indexes converge to the NAIVE one. ITEMPART outperforms the other three indexes, while LAZYORDER has higher execution time than SIMPLE, despite the early stopping condition defined. The relatively good

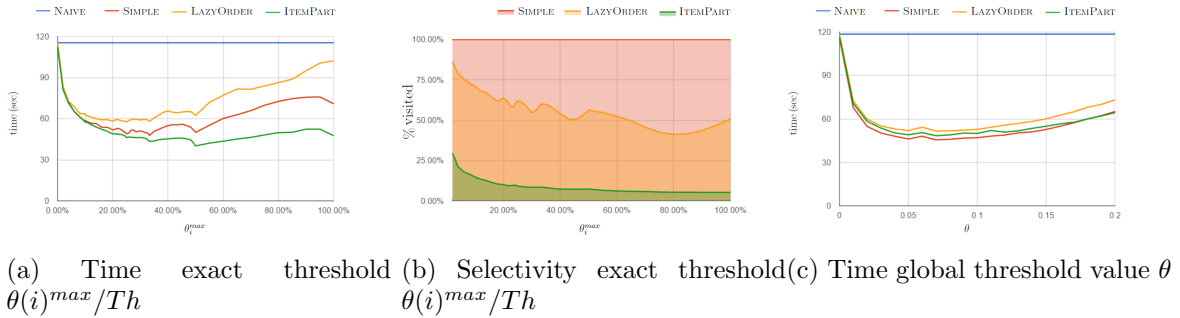


Figure 4: Exact and Global Threshold

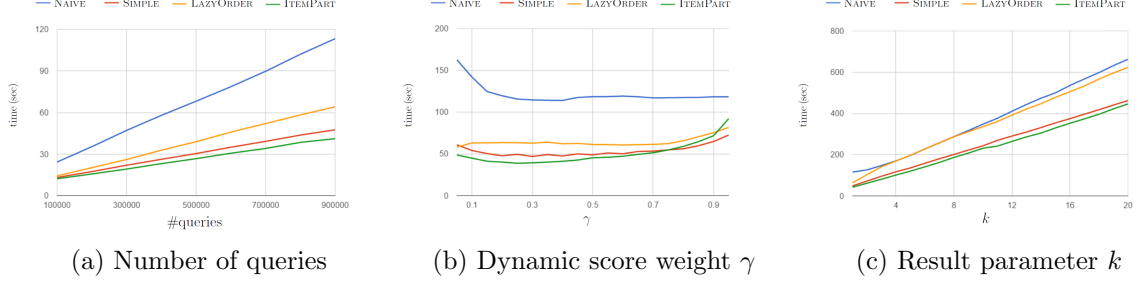


Figure 5: Scalability

performance of SIMPLE is attributed to the low maintenance cost (no re-orderings) and the use of a dynamic vector data structure, which guarantees fast access, insertions and deletions. These factors compensate for the lack of an early stopping condition. LAZYORDER and ITEM PART use much “heavier” data structures and need to prune a large portion of the candidates lists in order to outperform SIMPLE.

We can also observe a particular pattern in this plot: execution time exhibits local minima for $\theta(i)$ values which are fractions of $\theta(i)^{max}$, i.e., 100%, 50%, 33%, 25% etc. To understand this form, consider for example the case of $\theta(i) = 0.5 \cdot \theta(i)^{max}$. On the first evaluation of any given item i , the list of candidates with a difference up to $\theta(i)$ is computed. Since $\theta(i)$ corresponds to half the maximal value of the aggregated item score, the item will be re-evaluated through the Item Handler a second time, and there will be no need for a third evaluation. When a higher value is chosen, e.g., $0.6 \cdot \theta(i)^{max}$, two evaluations in the Item Handler would also be required, with the difference that some already known candidates would be retrieved again. Hence, an additional cost incurs to a) compute the redundant candidates and b) to filter out the probably higher number of false positives on event matching in the Event Handler.

Figure 4b shows the average percentage of lists visited, until the stopping condition becomes true. The size of the candidate lists is the same for all algorithms for the same $\theta(i)$ threshold. SIMPLE always visits 100% of the lists due to the lack of a stopping condition. We can observe that ITEM PART has a 20% smaller execution time (Figure 4a) while visiting in average only 5% of the candidate lists.

Global threshold θ : In this experiment we assign *the same θ value to all items* without using any knowledge of the maximal dynamic score per item. Figure 4c shows, for each *absolute value* θ assigned to all items, the time required to evaluate the whole dataset (DS5). We can observe that as the value of $\theta(i)$ increases from 0, there is a quick improvement in the performance of the indexes SIMPLE, LAZYORDER and ITEM PART while after exceeding the optimal $\theta(i)$ point, it deteriorates with a smaller slope: for very small values of θ , the lists become frequently obsolete (when the $\theta(i)$ -condition no longer holds) and a large number of incoming events need to be evaluated in the Item Handler. This explains the first phase where execution time decreases. As values of θ become much bigger, the lists become longer: computing the candidates becomes more costly and more false positives are likely to appear. Unlike in the previous experiment the three indexes exhibit similar performance with no more than 5% of difference in execution time. This behavior indicates that the stopping condition for both LAZYORDER and ITEM PART fails to prune as much query candidates as in the previous experiment. This is attributed to the arbitrary assignment of the same θ value to all items. For some items this value can be large w.r.t. $\theta(i)^{max}$ and the algorithms thus spend unnecessary time for finding candidate queries which will never be used. On the other hand, a small value of the $\theta(i)$ w.r.t. $\theta(i)^{max}$ means that there will be too many costly evaluations of events in the Item Handler.

Scalability and throughput: In this experiment, we are interested in the scalability of our indexes w.r.t. the number of continuous queries stored in the system (index creation time is excluded from our measurements). As we can see in Figure 5a all implementations scale linearly with the number of continuous queries. ITEM PART scales better: over 100,000 queries it requires 50% of the corresponding time for NAIVE while over 900,000 it only requires 36%. The low slope of ITEM PART indicates the good performance of its stopping condition to prune more candidate queries over increasing list sizes. In terms of throughput, over 900,000 continuous queries ITEM PART is able to serve using a single CPU core an average of 3.2 million items (tweets) or events (retweets) per minute which is one order of magnitude more than the number of tweets Twitter actually receives. Over a total of 100,000 stored continuous queries the throughput would go up to 10.9 million items and events per minute.

Item/event score weight γ : In this experiment, we vary the weight of the dynamic score (aggregating event scores) to the total score of an item (i.e., the γ parameter in Equation 1). Recall that when γ is small, each arriving event has a minimal impact on the total item score and only a small number of queries will be updated by the corresponding target items of incoming events. As shown in Figure 5b, NAIVE performance improves for values of γ up to 0.2 and then exhibits a constant behavior. On the contrary, the three other indexes require a higher execution time for greater γ values. In fact, the more γ becomes important, the more the initial static score between queries and items becomes obsolete. As score changes due to any single event become more significant, the candidate lists computed on item matching are soon invalidated given that minimal score of indexed candidate queries has changed. This leads to an increasing number of false positives and explains the increase in execution time. For realistic γ values in $[0.1, 0.5]$ the performance difference of any of these three indexes to their optimal value is of less than 10%.

Result size k : Figure 5c shows the performance of the four indexes over different values of the k parameter. Higher k values result in lower values of minimal scores for the stored queries and consequently increase the number of item updates received by the queries. As we can see in Figure 5c all indexes scale linearly on the value of k and that ITEM PART and SIMPLE exhibit a better performance of about 20% than the other two indexes when the $k = 20$.

Events per item: In this experiment we measure the overall performance of our indexes over the three datasets DS1, DS5 and DS10 (see Table 2) that feature a different average number of events each of the items receives: DS1 has an average of only 1.29 events per item while DS10 has 19.60. Figure 6 shows per dataset the *average* execution time for matching a single item or event along with the average number of updates (#updates) implied. We can observe that for all indexes execution time increases proportionally w.r.t. the number of updates. Additionally, the performance of the R^2TS algorithm for the three indexes (SIMPLE, LAZYORDER and ITEM PART) is better than NAIVE as the number of events per items increases. Over the DS1 dataset (with the smallest number of events/item), ITEM PART, which is the best performing index, needs 68% of the time required by NAIVE, while over DS10 it only requires 29,8%.

Decay: In our final experiment in Figure 7, we measure the performance of all indexes over linear decay. The horizontal axis shows the time it would take for a score of 1.0 to decay to 0, i.e., the maximum expiration time of an “idle” item (an item receiving no further events). A general observation is that as score decay becomes faster, indexes performance become worse: fast decay leads to a high number of updates (see number of updates in Figure 7), which in turn, leads to a higher delay in matching items and events. More precisely, while SIMPLE exhibits a 20-30% overhead over NAIVE the performance of LAZYORDER and ITEM PART becomes worse than NAIVE even with a small decay factor. This behavior indicates that the ordering maintained by

these two indexes becomes very dynamic and the stopping condition employed, is not sufficient to overcome the high maintenance cost. SIMPLE on the other hand, does not require any order maintenance and achieves a faster item and event matching time.

Conclusions on the experiments: Our experiments demonstrate that the ITEM PART and SIMPLE solutions outperform the NAIVE and LAZY ORDER ones over all settings, with ITEM PART having a slight edge over SIMPLE of about 5%. Comparing the stopping conditions, ITEM PART manages to filter with the defined stopping condition a big percentage of the candidate lists and

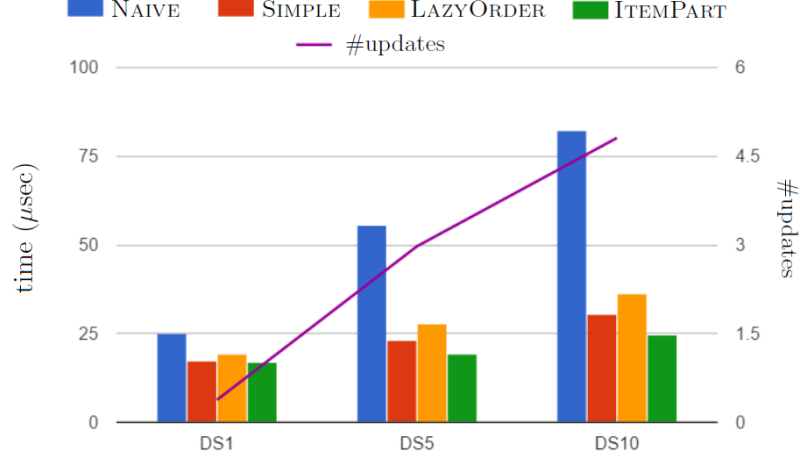


Figure 6: Scalability events per item

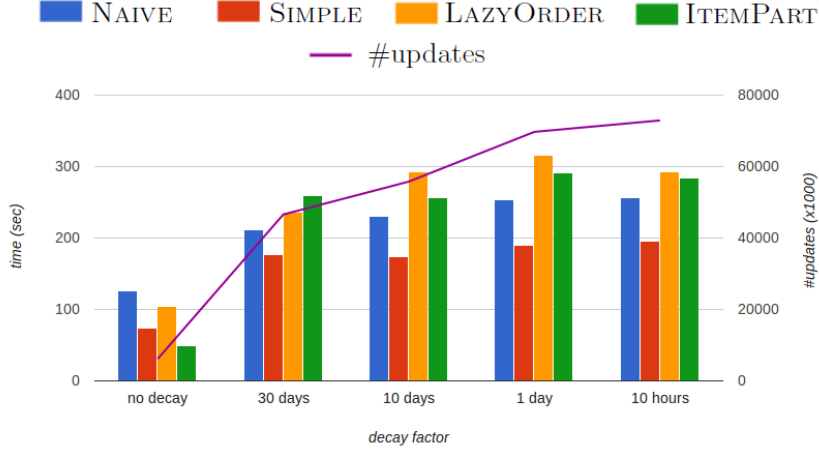


Figure 7: Scalability (linear) decay

on most cases only had to visit about 10% of the candidates before having correctly identified all updates and stopping the algorithm. However, the heavy data structures maintained for each item (an unsorted set of sorted lists) only allows ITEM PART to require from 50 to 35% the time used by the NAIVE index, which always visits all stored candidates. Our indexes perform better over highly dynamic environments with high numbers of events per published item.

6 Related work

Real-time search engines: Twitter Index (TI) was the first system proposed for real time search [3]. It relies on a general form of scoring functions combining linearly textual similarity of items to queries with (static and dynamic) query-independent parameters. A “*User PageRank*” authority measure is employed to estimate statically the importance on an item while its dynamic score is computed by counting tweet replies. A rational decay function is applied on the result to capture the freshness of information: $\mathcal{S}_{tot}(q, i)/\Delta\tau$, where $\mathcal{S}_{tot}(q, i)$ is the initial query-item score and $\Delta\tau$ is the time difference since the publication of the item. TI adopts a partial indexing approach that distinguishes frequent keyword queries from infrequent ones. For any new tweet, if it is relevant to at least one frequent query, it is inserted into an inverted index immediately; otherwise, it is inserted to an append-only log, the content of which is periodically flushed into the inverted index. However, user queries are evaluated only using the inverted index without looking at the log. The posting lists of this index are ordered on decreasing arrival time of tweets to privilege access to recent tweets even without sufficient consideration of their relevance to queries.

Deeper insights on how Twitter’s Search actually works have been published in [2]. To efficiently index all incoming tweets, Earlybird uses compression techniques for tweets’ contents and for long posting lists, multi-threaded algorithms and resources allocation optimizations. It relies on an append-only inverted index where each posting list stores tweets sorted by arrival time. This order enables to consider an effective stopping condition: when, during the traversal of the posting list, k items with a similarity score greater than a value s_{min} have been found, the scanning of the posting lists can stop *iff* s_{min} is greater than the maximal value of any other available tweet given that it is published exactly the same time as the one currently being checked. The estimation of this maximal value is based on the score decay function. Several works that followed [31, 22, 19] proposed improved versions of the inverted index for increasing the accuracy of results returned to real-time search or for supporting various forms of analytic queries without considering continuous query evaluation issues.

Top-k publish/subscribe on text streams: The Incremental Threshold (IT) algorithm [25] was the first work that introduced continuous top- k queries with sliding information expiration windows to account for information recency. It maintained two inverted indexes: one for queries and the other for items. The latter contains only the most recent items belonging to the current window and its posting lists are sorted by decreasing weight of the terms assigned to represent the items. In the former the posting lists are sorted in decreasing order of a value $\theta_{q,t}$, where t is the term of the posting list and q the continuous query that contains it. IT relies on the Fagin’s Threshold Algorithm (TA) to evaluate queries and on an early stopping condition to maintain their result lists. It continuously updates the inverted index on item publication and expiration and thus incurs a high system overhead. This limitation is addressed by the COL-filter algorithm [15] maintaining only an inverted index of queries with a sorted posting list for each query term. The sorting criteria takes account for the term weight and the minimum query score and allows to define a necessary and sufficient stopping condition for monotonic, homogeneous scoring functions such as cosine or Okapi-25. [15] maintains posting lists where queries are sorted according to the minimum top- k scores and items expire after some fixed time period (window based decay). Minimum top- k scores can change frequently and induce an important resorting overhead. This body of work cannot address continuous top- k queries beyond query-dependent item scores (i.e., content similarity). Efficient stopping conditions over a two-dimensional representation of queries featuring both a query-dependent and query-independent scores (with time decay) have been firstly proposed in [27]. In this paper, we leverage the item matching algorithm implementing the Item Handler for static scores with

time decay and propose several alternative index structures for implementing event matching in the Event Handler (see Figure 2) that accounts for the dynamic score of items due to social feedback.

Finally, [26] considers the problem of annotating real-time news stories with social posts (tweets). The proposed solution adopts a top- k publish-subscribe approach to accommodate high rate of both pageviews (millions to billions a day) and incoming tweets (more than 100 millions a day). Compared to our setting, stories are considered as continuous queries over a stream of tweets while pageviews are used to dynamically compile the top- k annotations of a viewed page (rather than for scoring the news). The published items (e.g., tweets) do not have a fixed expiration time. Instead, time is a part of the relevance score, which decays as time passes. Older items retire from the top- k only when new items that score higher arrive. This work adapts, optimizes and compares popular families of IR algorithms without addressing dynamic scoring aspects of top- k continuous queries beyond information recency, namely TAAT (term-at-a-time) where document/tweet scores are computed concurrently one term at a time, and DAAT (document-at-a-time) where the score of each document is computed separately.

Top- k publish/subscribe on spatio-textual streams: AP (Adaptive spatial-textual Partition)-Trees [30] aim to efficiently serve spatial-keyword subscriptions in location-aware publish/subscribe applications (e.g. registered users interested in local events). They adaptively group user subscriptions using keyword and spatial partitions where the recursive partitioning is guided by a cost model. As users update their positions, subscriptions are continuously *moving*. Furthermore, [29] investigates the real-time top- k filtering problem over streaming information. The goal is to continuously maintain the top- k most relevant geo-textual messages (e.g. geo-tagged tweets) for a large number of spatial-keyword subscriptions simultaneously. Dynamicity in this body of work is more related to the queries rather than to the content and its relevance score. Opposite to our work, user feedback is not considered. [6] presents SOPS (Spatial-Keyword Publish/Subscribe System) for efficiently processing spatial-keyword continuous queries. SOPS supports Boolean Range Continuous (BRC) queries (Boolean keyword expressions over a spatial region) and Temporal Spatial-Keyword Top- k Continuous (TaSK) query geo-textual object streams. Finally, [5] proposes a temporal publish/subscribe system considering both spatial and keyword factors. However, the top- k matching semantics in these systems is different from ours (i.e., boolean filtering).

In our work, online user feedback is part of the ranking score of items, which renders existing spatial-keyword solutions inapplicable, while challenges index structures and matching algorithms for top- k continuous query evaluation.

7 Conclusions

To support real-time search with user feedback, we have introduced a new class of continuous top- k queries featuring complex dynamic scores. The main problem we solved in this context, is the search of queries that need to update their result lists when new information items and user events affecting their scores, arrive. To accommodate high arrival rates of both items and events, we have proposed three general categories of in-memory indexes and heuristic-based variations for storing the query candidates and retrieving the result updates triggered by events. Using an analytic model we have theoretically proven efficient early stopping conditions avoiding visiting all candidate queries for each item. Our experiments validated the good performance of our optimized matching algorithm. All three indexes for candidate maintenance (SIMPLE, LAZYORDER and ITEM PART) achieve a high throughput of items and events, with ITEM PART being able to handle 3.2 million events per minute over 900 thousand stored continuous queries. This performance has been achieved using a centralized single-threaded implementation of our

algorithms. The usage of posting-lists as basic data structures for filtering events opens a number of opportunities for parallelization. The main challenge in this direction concerns the reordering cost of the ordered solutions (LAZYORDER and ITEM PART) with early stopping conditions. One direction of future work is to consider workload-oriented parallelization by identifying independent item clusters (for example, with disjoint query keywords). Another direction is to extend the unordered solution SIMPLE, which is certainly a good candidate for a first parallel implementation. Finally, we intend to study richer top- k query settings with random negative feedback scores. The efficiency of our solution is strongly based on the monotonicity of the scoring function and adding negative score might need the exploration of completely different multi-dimensional indexing approaches.

References

- [1] A. Alkhouli, D. Vodislav, and B. Borzic. Continuous top- k processing of social network information streams: a vision. In *In ISIP 2014 post proceedings*, Springer 2015, 2015.
- [2] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at twitter. In *IEEE 28th International Conference on Data Engineering (ICDE)*, pages 1360–1369, April 2012.
- [3] C. Chen, F. Li, B. C. Ooi, and S. Wu. Ti: An efficient indexing mechanism for real-time search on tweets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’11, pages 649–660. ACM, 2011.
- [4] L. Chen, G. Cong, and X. Cao. An efficient query indexing mechanism for filtering geo-textual data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’13, pages 749–760, New York, NY, USA, 2013. ACM.
- [5] L. Chen, G. Cong, X. Cao, and K. Tan. Temporal spatial-keyword top- k publish/subscribe. In *31st IEEE International Conference on Data Engineering*, pages 255–266, Seoul, South Korea, 2015.
- [6] L. Chen, Y. Cui, G. Cong, and X. Cao. Sops: A system for efficient processing of spatial-keyword publish/subscribe. *Proc. VLDB Endow.*, 7(13):1601–1604, Aug. 2014.
- [7] J. Cho and H. Garcia-Molina. Effective page refresh policies for web crawlers. *ACM Trans. Database Syst.*, 28(4):390–426, 2003.
- [8] V. Christophides and T. Palpanas. Report on the first international workshop on personal data analytics in the internet of things (pda@iot 2014). *SIGMOD Rec.*, 44(1):52–55, May 2015.
- [9] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu. Forward decay: A practical time decay model for streaming systems. In *IEEE International Conference on Data Engineering (ICDE’09)*, pages 138–149, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
- [10] A. Dong, R. Zhang, P. Kolari, J. Bai, F. Diaz, Y. Chang, Z. Zheng, and H. Zha. Time is of the essence: Improving recency ranking using twitter data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW ’10, pages 331–340. ACM, 2010.
- [11] Y. Duan, L. Jiang, T. Qin, M. Zhou, and H.-Y. Shum. An empirical study on learning to rank of tweets. In *Proceedings of the 23rd International Conference on Computational*

- Linguistics*, COLING '10, pages 295–303, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.
- [12] M. Grinev, M. P. Grineva, M. Hentschel, and D. Kossmann. Analytics for the realtime web. *PVLDB*, 4(12):1391–1394, 2011.
 - [13] L. Guo, D. Zhang, G. Li, K.-L. Tan, and Z. Bao. Location-aware pub/sub system: When continuous moving queries meet dynamic event streams. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 843–857, New York, NY, USA, 2015. ACM.
 - [14] A. Gupta and P. Kumaraguru. Credibility ranking of tweets during high impact events. In *Proceedings of the 1st Workshop on Privacy and Security in Online Social Media*, PSOSM '12, pages 2:2–2:8. ACM, 2012.
 - [15] P. Haghani, S. Michel, and K. Aberer. The gist of everything new: personalized top-k processing over web 2.0 streams. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM'10)*, pages 489–498. ACM, 2010.
 - [16] R. Horincar, B. Amann, and T. Artieres. Online change estimation models for dynamic web resources. In M. Brambilla, T. Tokuda, and R. Tolksdorf, editors, *Web Engineering*, volume 7387 of *Lecture Notes in Computer Science*, pages 395–410. Springer Berlin Heidelberg, 2012.
 - [17] A. Khodaei and C. Shahabi. Social-textual search and ranking. In *Proceedings of the First International Workshop on Crowdsourcing Web Search, Lyon, France, April 17, 2012*, pages 3–8, 2012.
 - [18] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.
 - [19] Y. Li, Z. Bao, G. Li, and K.-L. Tan. Real time personalized search on social networks. In *Data Engineering, 2015 IEEE 31st International Conference on*, pages 639–650, April 2015.
 - [20] J. Lin and G. Mishne. A study of "churn" in tweets and real-time search queries. In *Proceedings of the Sixth International Conference on Weblogs and Social Media (ICSWM)*, Dublin, Ireland, 2012.
 - [21] J. Liu, P. Dolan, and E. R. Pedersen. Personalized news recommendation based on click behavior. In *Proceeding of the 14th international conference on Intelligent user interfaces (IUI'10)*, IUI '10, pages 31–40. ACM, 2010.
 - [22] A. Magdy, M. Mokbel, S. Elnikety, S. Nath, and Y. He. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. In *Data Engineering, 2014 IEEE 30th International Conference on*, pages 172–183, March 2014.
 - [23] X. Mao and W. Chen. A method for ranking news sources, topics and articles. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 4, pages 170–174, Apr. 2010.
 - [24] Y. Miao, C. Li, L. Yang, L. Zhao, and M. Gu. Evaluating importance of websites on news topics. In *PRICAI 2010: Trends in Artificial Intelligence*, volume 6230 of *LNCS*, pages 182–193, 2010.

- [25] K. Mouratidis and H. Pang. Efficient evaluation of continuous text search queries. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23:1469–1482, 2011.
- [26] A. Shraer, M. Gurevich, M. Fontoura, and V. Josifovski. Top-k publish-subscribe for social annotation of news. *Proc. VLDB Endow.*, 6(6):385–396, Apr. 2013.
- [27] N. Vouzoukidou, B. Amann, and V. Christophides. Processing continuous text queries featuring non-homogeneous scoring functions. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management, CIKM '12*, pages 1065–1074. ACM, 2012.
- [28] C. Wang, M. Zhang, L. Ru, and S. Ma. Automatic online news topic ranking using media focus and user attention based on aging theory. In *Proceeding of the 17th ACM Conference on Information and Knowledge Management (CIKM'08)*, pages 1033–1042. ACM, 2008.
- [29] X. Wang, Y. Zhang, W. Zhang, X. Lin, and Z. Huang. Skype: Top-k spatial-keyword publish/subscribe over sliding window. *Proc. VLDB Endow.*, 9(7):588–599, Mar. 2016.
- [30] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. Ap-tree: Efficiently support location-aware publish / subscribe. *The VLDB Journal*, 24(6):823–848, 2015.
- [31] L. Wu, W. Lin, X. Xiao, and Y. Xu. Lsii: An indexing structure for exact real-time search on microblogs. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 482–493, 2013.